Computers Math. Applic. Vol. 22, No. 10, pp. 49–63, 1991
Printed in Great Britain.

0097-4943/91 $3.00 + 0.00

COMPUTING STRUCTURE: SETS, STRUCTURES, AND INVARIANTS IN LISP

JOHN A. WINNIE
Department of History and Philosophy of Science, Indiana University
Goodbody Hall 130, Bloomington, IN 47405, U.S.A.

(Received March 1991)

Abstract—Sets which may contain other sets as members are defined as a data type in *Scheme*, a dialect of LISP. Using the sets so-defined, the standard set-theoretical operations of union, intersection, powerset, etc. are developed. Drawing upon the theory of structures presented in [1], set-theoretical structures are defined and procedures for computing their automorphisms (symmetries) and invariants are developed.

1. INTRODUCTION

The structures most commonly associated with computation are collections of variables of various types, grouped together under a single label for easy access. There is, however, another kind of "structure," perhaps more familiar to logicians than programmers, but well worth considering by those who work with symbolic representations. These are the set-theoretical structures: the various spaces, algebras, and relational systems so common in contemporary logic and mathematics. The theory of groups, for example, studies ordered pairs of the form $\langle G, \circ \rangle$, where G is a (non-empty) set of objects and $\circ$ is a binary operation from $G \times G$ to G that behaves like functional composition. A metric geometry is also a pair $\langle P, d \rangle$, but this time d is a metric on P, a function from $P \times P$ to the real numbers that acts like a measure of distance between the "points" in the set P. A possible model (or "relational system") of predicate logic is a set paired with a sequence of relations defined on that set. These examples all illustrate the same fundamental idea: a structure is considered to be a non-empty set, or "universe," paired with one or more set-theoretical operations or relations defined on that universe. The investigation of these structures treats these pairs as objects in their own right, and studies them in detail by using algebraic, analytical, or set-theoretical techniques.

One such technique is to survey the structure-preserving mappings from one structure to another, and often, from a structure to itself. For example, consider all of the one-one mappings of the universe of a structure onto itself, in other words, the permutations of that universe. Those permutations that leave the entire structure intact are called *automorphisms* or *symmetries* of the structure. An *invariant* of a structure is a set (defined on the universe of the structure) that is preserved by every automorphism of the structure. The defining operations or relations of a structure will always be invariants, but there will be other invariants as well. Intuitively, invariants are "objective" features of a structure (see [2] for more details and a number of charming examples).[1]

An important property of the automorphisms of a structure is that they form a group when functional composition is taken as the group operation. Often, this automorphism group provides important information about the underlying structure under investigation; more generally,

I am indebted to Burke Townsend (University of Montana) for many helpful discussions early on, and Linda Wessels (Indiana University) for many later discussions. George Springer (Indiana University) has been of help with some of the ins and outs of *Scheme*.

[1] Automorphisms and invariants will be defined more precisely and discussed in more detail in Section 6.

the investigation of all mappings from a given structure to other structures (homomorphisms, isomorphisms, embeddings, etc.) has become a major tool in the study of these structures.

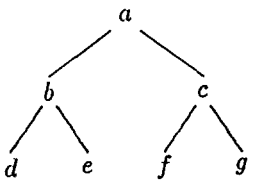
The purpose of this paper is to provide a conceptual (and working) framework for the representation of structures and the computation of their automorphisms and invariants. Since structures, in this sense, are simply complicated kinds of sets, the fundamental tool for such representations will be sets—not merely sets of simple elements or atoms, but sets in the full-blooded sense: sets that may contain other sets in turn. The language I have chosen for this purpose is *Scheme*, although the account given here should transfer easily to other equally powerful dialects of LISP.

Since my main concern has been to lay out the fundamentals of the subject as clearly as I can, when forced to choose between conceptual clarity and computational costs, I have almost always opted for the former. On the few occasions when I have made some concessions to computational costs, I point this out. The technical basis for the theory of structures implemented here can be found in [1], especially Section 3, where there are more details, including the appropriate theorems and their proofs.

2. SETS, LISTS, AND STRUCTURES

The principal differences between lists and sets are as obvious as they are important: Lists order their members and notice their repetition; sets do neither. Thus, the collections $(a\,a\,b\,c)$, $(a\,b\,c)$, and $(b\,a\,c)$, while distinct as lists, are the same as sets. Sets and lists alike, however, gain most of their power to represent complex structures from their ability to contain members of their own kind. In the case of lists, association lists and property lists are familiar examples. In the case of sets, the well-known construction of the various number systems from the empty set is a striking illustration of the conceptual power of hierarchies of sets.

With their built in order, lists allow complex structures to be constructed rapidly and represented concisely; sets, however, have the upper hand when it comes to representing the structural heart of a situation, when the aim is to depict only the structural essentials. Take, for example, the case of the *unordered* binary tree pictured below.[2]



One way of representing this tree is by a list that pairs each of its nodes with the list of its offspring. In this case, the result would be

$$((a\,(b\,c))\,(b\,(d\,e))\,(c\,(f\,g))).$$

The same tree, however, can also be represented by using sets and pairing each node with the set of its offspring:

$$\{\{a,\{b,c\}\},\{b,\{d,e\}\},\{c,\{f,g\}\}\}.$$

In general, the "naked" member of each set represents a node of the tree, which is then paired with the set of its offspring.

Although both representations capture the structure of the tree and permit the definition of methods for its search, there is an important conceptual difference between the two representations. The list representation adds structure (and, hence, information) gratuitously. An unordered tree is defined by its nodes and their offspring, but does not provide a "left-to-right" order of those offspring. The list representation, by virtue of its built-in order, hides the symmetries of the tree, whereas the set representation permits us to extract them from the representation

[2]Knuth [3, pp. 371–372] calls unordered trees "oriented" trees. When the order of the offspring of a tree's nodes *is* relevant, the tree is said to be "ordered." In that case, the list representation represents the (rigid) structure of the tree perfectly.

itself. Thus, the permutation

$$\begin{pmatrix} a & b & c & d & e & f & g \\ a & c & b & f & g & d & e \end{pmatrix}$$

is an automorphism or symmetry of the set representation of the tree, since it maps the representation to itself. Only the identity permutation, however, maps the list representation of the tree to itself; unlike the structure it represents, the list representation has no non-trivial symmetries.

In a word, lists are rigid. They totally individuate each item right down to each of its occurrences. For computational bookkeeping, this feature of lists is invaluable; but when the problem is to represent structures with important symmetries, it is sets, not lists, that can provide the required flexibility. When order is an unimportant or accidental feature of a structure, sets may be used *simpliciter*; when—as in the previous example—some aspect of a structure requires order, then more complex sets such as pairs, ordered pairs, or sequences may be invoked.

Such are the advantages of having sets and their attendant representational power available in LISP; but how, exactly, is this to be done? The basic difficulty is that lists are *ordered* collections; from the set-theoretical standpoint, when we begin with lists, we are beginning with a surplus of structure. If sets and their relations are to be defined within LISP, we must somehow or other shed some of that structure and abstract from a list's constant concern for the order and possible repetition of its members.

The problem of defining sets has been handled in the LISP literature in a number of ways. What seems to be the standard approach, however, begins by defining a set as a concise list, a list without repetitions.[3] Set membership is then simply taken to be the standard LISP 'member' relation, and one set is a subset of another just in case every member of the first is a member of the second. Two sets are set-equal if each is a subset of the other. Set-theoretical union, intersection, difference, etc. are then defined in the expected ways, taking care to avoid creating duplicates along the way.

While this approach is fine for simple sets (sets of non-sets), it does not provide us with full-fledged sets that can contain their own kind as members and still have set-equality come out right. The problem stems from beginning with the LISP 'member' relation or its equivalent; once we start this way, we are locked in on an exclusively 'top-level' treatment of sets. This is an important point, so it is worth seeing clearly.

Consider the "sets"

$$A = ((a\,b)\,c)$$

and

$$B = ((b\,a)\,c).$$

Since the list $(a\,b)$ is a member of A but not a member of B, then according to the standard approach, A and B are not the same set: they have different members. The difficulty here is that list-membership and set-membership only coincide when the elements being considered are atoms. The set $(a\,b)$ is a set-member or element of set B because it is the same set as $(b\,a)$, but failing this knowledge, A and B must be judged to be different sets. Such is the inevitable consequence of starting with LISP's built-in member relation. If we cannot start with membership, however, then where can we start?

The following account begins by defining *set-equality*, rather than set-membership. With equality on hand, an element of a set may be defined simply as an item that has the set-equality relation to some member (in the LISP sense) of the set. At this point, the subset relation, intersection, union, and the other standard relations and operations on sets may be defined in the obvious ways. So if we can define set-equality, all the rest comes naturally; but how do we manage to do that?

[3] See [4, p. 110ff; 5, p. 123ff; and 6, p. 73]. For an approach to sets of sets compatible with that presented here and derived in this respect from an earlier version of this paper, see the recent text [7, Chapter 8].

Fortunately, set theory itself provides the answer to this question. Sets are identical, according to the axiom of extensionality, when they contain the same members. These members, in turn, are identical when their members are identical, and so it goes. The situation would seem to invite recursion, provided that all goes well when the process comes to ground; at that point, we shall require entities whose identity relation is both procedurally available in LISP and set-theoretically acceptable. Fortunately, the individuals out of which lists (and thus sets) can be built are just the atoms (in *Scheme*, the empty list and strings are hereby included), and between atoms the standard LISP **eq?** relation (or *Scheme*'s **eqv?** relation) is just the ticket. Sets, then, will be identical when they have identical members as extensionality demands, and ultimately their identity will be recursively grounded in the standard LISP equality of the atoms from which they are built up.[4]

In the next section, a number of useful logical operations will be defined for later use. Following this, sets and the fundamental set-theoretical operations are defined. In Section 5, an account of functions and the images of sets under a function is developed, and in the next section, structures are defined, along with procedures for computing their automorphisms and invariants. Finally, I present a few examples and offer some suggestions for extensions and improvements.

3. LOGICAL OPERATORS

In the following developments, it will be useful to have certain logical operations on hand. In all but a few cases, the definition of the procedure should be obvious (see Listing I for the procedures of this section).

The first three functions, **some?**, **all?**, and **none?**, take a property (a single argument predicate) and a list as arguments. **Some?** determines whether or not at least one member of the list has the property supplied, **all?** checks on whether or not every member of the list has the property, and **none?**, as might be expected, asks if no member of the list has the property. All three operators are genuine predicates and thus return a "true" when satisfied, and the empty list '() otherwise.[5]

Using the **some?** and **all?** predicates, it is now an easy matter to define more complex logical predicates, such as **one-every?**. This function takes three arguments: an arbitrary expression e, a two-placed predicate (binary relation), and a list. If the item e has the relation to every item in the list, then "true" is returned; otherwise the function returns the empty list. For example, (**one-every?** 12 > '(2 4 6)) returns true, whereas (**one-every?** 12 > '(8 10 12)) returns the empty list '(). The explanations of the next three functions, **every-one?**, **one-some?**, and **every-some?** are similar: the first determines whether or not every item in a list has the relation to a given item, the second is satisfied when a given item has the relation to some item in a given list, and the last is satisfied when every item in the first list has the relation to some item in the second.

4. SETS AND SET-THEORETICAL OPERATIONS

Before proceeding to sets proper, a few words on general methodology. It follows from my preceding remarks about the set-equality relation that repetitions within a set are no problem—at least in principle. A well-behaved set-equality relation will ignore repetitions (and order) and so equate all and only the appropriate lists. Thus, there is no need to define "sets" as such, just the standard set-theoretical relations and operations. Lists that behave like sets with respect to these relations and operations *are* sets, at least for all scientific purposes.[6] Nevertheless, repetitions within a set are computationally very expensive, so at the very least they ought not to be encouraged, much less nurtured. The function **collapse**, defined at the end of this section, serves to eliminate duplicates with respect to the set-equality relation at all levels within a list.

[4] I am, in effect, assuming what set-theorists call "the axiom of regularity." In the case of finite set theory with individuals—which is what we have here—the use of this axiom should not be controversial.

[5] It might be thought that it would be more convenient not to make **some?** a genuine predicate and have it return the first item in the list having the property in question. However, **some?** may be applied to sets of sets—and, hence, lists of lists—that may contain the empty list. In a case like (**some?** **null?** '(*ab*())), we want the result to be "true," not '()!

[6] This point about identity and definitions is discussed in more detail in [8, Sections 24 and 53].

In addition, all set-theoretical operations will be defined so that if they are initially supplied with concise sets (lists without repetitions) the resulting set is also concise.[7]

We begin with the set-equality relation, here called "**same-set?**". (See Listing II for a complete list of the functions discussed in this section.)

```
(define same-set?
 (lambda (l m)
  (cond
   ((eqv? l m) t)
   ((or (atom? l) (atom? m)) nil)
   (else (and (subset? l m) (subset? m l))))))

(define subset?
 (lambda (l m)
  (every-some? l same-set? m)))
```

These definitions, notice, are indirectly or mutually recursive. First, atoms that the LISP **eqv?** predicate counts as the same are to be the same set.[8] Next, if two items are not **eqv?**, then if either is an atom, they cannot be the same set. And finally, extensionality: set-equality is to hold just in case every member of the first is the same set as some member of the second (the first is a subset of the second), and conversely. The **subset?** relation is defined as holding between sets l and m just in case every item in l is the same set as some item in m.

Since these definitions are so fundamental to everything that follows, it is probably a good idea to see how they handle a simple example. Consider the (same) sets

$$A = ((a\,b)\,c)\;\; B = (c\,(b\,a)).$$

As the pair of definitions is followed through, the first serious question is whether or not A is a subset of B. This, in turn, requires that $(a\,b)$ is the **same-set?** as either c or $(b\,a)$, and the same, of course, must be true for the other member of set A, namely, c. Since c is an atom, that case is settled: c is the **same-set?** as c, since c is **eqv?** to c. Now back to set $(a\,b)$. It cannot be the **same-set?** as c, by the second clause of the definition of **same-set?**. Hence, we are left comparing $(a\,b)$ to $(b\,a)$. Once again, since neither are atoms, if they are to be the same set, $(a\,b)$ must be a subset of $(b\,a)$. This, in turn, requires the atom a to be the same set as some item in $(b\,a)$—which it is, and so it goes. This much should be enough to see how the pair of definitions work; eventually, all comparisons will come to ground in atoms, where the standard LISP **eqv?** predicate takes over.

Now that we have the set-equality relation on hand, the set-membership relation, here called "**element?**", is immediately forthcoming: An item e is an **element?** of a set l if and only if e is the **same-set?** as some item in l.

The definition of the powerset of a set—the sets of all its subsets—is a bit more complicated. Starting with the set $(a\,b\,c)$, say, we want to obtain

$$(()\,(a)\,(b)\,(c)\,(a\,b)\,(a\,c)\,(b\,c)\,(a\,b\,c)).$$

The basic idea of the **powerset** procedure's definition is that, once you have the powerset of the **cdr** of a list, in this example,

$$(()\,(b)\,(c)\,(b\,c)),$$

the powerset of the entire list may be obtained by inserting its **car**—in this case the atom a—into each of its members and then appending the result to the original.

The cardinal number of a set is obtained by simply counting its members, taking care to discard **same-set?** members along the way. **Set-union**, **set-intersection**, and **set-difference** are now defined in the standard ways, taking care—especially in the first case—that duplicates are not

[7] For a discussion of the computational costs of defining set-theoretical operations, see [4, p. 111ff].

[8] Atoms are hereby counted as "sets," but this does no real harm, and makes for over-all smoothness.

generated. Corresponding to **set-union** and **set-intersection** are their counterpart operations on families of sets, called **union-of-sets** and **intersection-of-sets**; when applied to a set of sets (a family of sets) l, they yield the union and intersection, respectively, of the sets contained in l. Their definitions are recursive and straightforward.

The final set-theoretical operation in this group is called "**separate**," after Zermelo's axiom of separation. When supplied with a one-placed predicate and a set, **separate** returns all those items in the set that satisfy the predicate. For example, (**separate even?** '(1 2 3 4)) returns (2 4).

The procedure **collapse** finishes this group. As I mentioned earlier, since repetitions in a set can be computationally expensive, it is convenient to have a way to eliminate them when we care to. The function **collapse** does just this; when applied to a list, **collapse** purges it of set-theoretical duplicates at all levels. It works by using double recursion to throw out all equivalents with respect to the **same-set?** relation.

5. ORDERED PAIRS, FUNCTIONS, AND IMAGES

A full theory of structure, as we have seen in the introduction, requires access to various mappings of structures and the images of sets under these mappings. In short, we need functions as first-class set-theoretical objects, both for the construction of structures and their exploration. In standard set-theory, functions are sets of ordered-pairs, and so shall they be here.

Ordered pairs may be defined in a number of ways; here they will be treated in the Kuratowski manner: the ordered pair $\langle a, b\rangle$ is defined as the set $\{\{a\}, \{a, b\}\}$. It turns out that if the pairs $\langle a, b\rangle$ and $\langle c, d\rangle$, when so-defined, are in the **same-set?** relation, we must have (**same-set?** a c) and (**same-set?** b d) true, regardless of the levels of the sets involved. Given an ordered pair, the **first-of-pair** is any member of its intersection; extracting the second member, however, is not so straight-forward. The complication is that the pair may be degenerate (as in $\langle a, a\rangle$) or may contain repetitions listed in an arbitrary order (as in $\langle a, b\rangle = ((a\, b\, a)\, (a\, a))$. The basic idea of the definition of **second-of-pair** is this: first, take the union of the pair and then "subtract" its intersection; if the result is empty, the pair must have been degenerate (i.e., of the form $\langle a, a\rangle$), so return the first member; otherwise, the result contains (only) the second member of the pair, so return that member.

The **cross-product** ($A \times B$) of two sets A and B is just the set of all ordered pairs with the first member in A and the second member in B. For example, the **cross-product** of ($a\, b\, c$) and (1 2) is simply the set $(\langle a, 1\rangle\, \langle a, 2\rangle\, \langle b, 1\rangle\, \langle b, 2\rangle\, \langle c, 1\rangle\, \langle c, 2\rangle)$.[9] The definition, using **mapcar** (or **map** in *Scheme*) is straightforward.

A (two-placed) *relation* is simply any set of ordered pairs, whereas a *function* is a relation of a special sort: no two pairs in a function may agree on their first members yet disagree on their second. In LISP, the natural way to represent functions is simply as a list of pair-lists, or an association list. From the set-theoretical standpoint, this will not work. We want functions to be "first-class" set-theoretical objects in their own right; this will allow them to be elements of the universe of some structure or other and, thus, become arguments of still other functions at a higher set-theoretical level.

Since functions so construed are somewhat complex objects, it is convenient to have an efficient way of producing them. The procedure **create-function** provides one such method. When fed the two lists ($a\, b\, c$) and (1 2 1), say, it returns the function $(\langle a, 1\rangle\, \langle b, 2\rangle\, \langle c, 3\rangle)$, making the appropriate ordered-pairs as it goes along. (To ensure that the result is a function—and not merely a relation—it suffices that the first list contain no set-theoretical duplicates.) The **domain** of a function is defined as the set of objects on which the function is well-defined. It is obtained by simply running through the function, extracting the first item of each pair, and collecting the results. Somewhat non-standardly, the **range** of a function is defined as the set of values taken on by the function over its domain.[10] The value of a function on an argument is exactly what

[9] In this example, and often from now on, the ordered pair of a and b will be written "$\langle a, b\rangle$," abbreviating the more cumbersome "$((a)\, (a\, b))$."

[10] The standard approach these days is to *assign* a range to a function when defining it, requiring only that the values of the function on its domain be elements of the assigned range. If the image of the function's domain is the

one would expect: the item that is paired with the argument in the function. If the argument supplied to the function is not an element of its domain, then an error message is returned. Given two functions f_1 and f_2, a third having the domain of f_2 may be obtained by composing them in the usual way. The procedure **compose** does this by running through the domain of f_2 and pairing each item i with $f_1(f_2(i))$. Unless the domain of function f_1 is a subset of the range of f_2, somewhere along the line an error message will be returned. A *sequence* is a special kind of function whose domain is some initial section of (or the entire set of) non-negative integers; the function $\{\langle 0,z\rangle,\langle 1,y\rangle,\langle 2,x\rangle\}$, for example, is a sequence with domain $\{0,1,2\}$. Since certain kinds of structures (e.g., relational systems in logic and model theory) are defined in terms of sequences, it is convenient to have a way to convert a list into a sequence when desired. The procedure **create-sequence** does this by running down the list, making ordered pairs of successively greater integers with each item, and collecting the result. For example, when the function **create-sequence** is supplied with the list $(a\,b\,c)$, the result is the set $(\langle 0,a\rangle\,\langle 1,b\rangle\,\langle 2,c\rangle)$.

The notion of the *image* of a set under a function is crucial to the following developments. The basic idea is that to obtain the image of a set A under a function f, take the set A apart until an element of the domain of f is found; then replace that element by its f-value. Continue in this way until there is nothing left to take apart. Suppose, for example, we define the function f on the alphabet $\{a,b,...,z\}$ so that f takes each letter to its successor (and takes z to a). Then the image of the set $((a\,b)\,c)$ under f is simply $((b\,c)\,d)$, and is obtained as illustrated below.

$$
\begin{array}{c}
f[((a\,b)\,c)] \\
\downarrow \\
(f[(a\,b)]\,f[c]) \\
\downarrow \\
((f(a)\,f(b))\,f(c)) \\
\downarrow \\
((b\,c)\,d)
\end{array}
$$

Once again, the situation suggests recursion. The image of a set may be obtained by taking the set of images of its elements, the set of images of the elements of the elements, and so on, down to the point where we encounter an item to which we may apply the function directly. Of course, somewhere on this downward journey we may meet an atom that is not in the domain of the function. In this case, the image is defined as simply the atom itself.[11] As a result, the image of a function is well-defined on *any* set, even if that set contains atoms that are not in the domain of the function.

Sometimes, when a function is applied to a set, the image of the set is identical to the original set. Of course, this will always be so when the function is the identity function, but this is not the only time this can happen. Consider the function h that reverses the usual alphabetical order; in other words, h is the permutation:

$$
\begin{pmatrix} a & b & c & \ldots & x & y & z \\ z & y & x & \ldots & c & b & a \end{pmatrix}.
$$

Then the image of the set $B = (b\,c\,x\,y)$ under h is just the set $(y\,x\,c\,b)$, which is the set B right back again. When the image of a function is the same set as we started with, the function is said to *preserve* the set. The predicate **preserves?** of Listing III defines this notion in the standard way. As we shall see in the next section, the automorphisms or symmetries of a structure are just those permutations of its universe that preserve its distinguished relations.

---

entire range, then it is *onto*—otherwise, *into*—the range. In the approach taken above, every function is trivially onto its range, but there is nothing to prevent attaching a wider set to a function if desired.

[11] This results in a (welcome) simplification of the account given in [1]. Notice that it remains true that the *value* of a function on an argument is undefined if the argument is not in its domain. Objects which are not in a function's domain play the role of logical dummies or markers when it comes to taking images. In this respect, they are like numbers.

6. SET-THEORETICAL STRUCTURES

A *structure*, in set-theory, is an ordered pair $\langle U, S\rangle$, where U is a non-empty set, and S is any set whatsoever. Typically, S is some set built up from elements of U and may involve numbers as "measures." I shall call the set S that "structures" the universe U the ***distinguished structor*** of that structure. For example, the ordered pair $\langle\{a,b,c,d\}, f\rangle$, where f is the function $\{\langle a,1\rangle,\langle b,2\rangle,\langle c,2\rangle,\langle d,3\rangle\}$, is a structure with universe $\{a,b,c,d\}$ and distinguished structor f. Often the distinguished structor of a structure is a sequence of sets, say relations or functions, defined on a given universe. The relational systems of model theory, groups, fields, etc. are all examples of a universe paired with a sequence of distinguished structors which, intuitively, "structure" the underlying set. (For more examples and details see the similar formulation in [1].) Since sequences are just sets in turn, however, the above definition of a structure as simply an ordered pair covers these cases.

An *automorphism* or *symmetry* of a structure is a permutation of its universe that preserves its distinguished structor. Thus the permutation

$$\begin{pmatrix} a & b & c & d \\ b & a & d & c \end{pmatrix}$$

changes the distinguished structor f of the previous example to

$$\{\langle b,1\rangle,\langle a,2\rangle,\langle d,2\rangle,\langle c,3\rangle\},$$

clearly not the same set f we had at the beginning. Hence, the above permutation *is not* an automorphism of this structure. On the other hand, the permutation

$$\begin{pmatrix} a & b & c & d \\ a & c & b & d \end{pmatrix}$$

changes the function f to $\{\langle a,1\rangle,\langle c,2\rangle,\langle b,2\rangle,\langle d,3\rangle\}$, and this, clearly, *is* the same set (function) f right back again. Hence, this last permutation is an automorphism of our structure. Obviously, the identity permutation of a structure's universe will always be an automorphism of that structure. An *invariant* of a structure is a set (typically a relation or operation on its universe) that is preserved by all of its automorphisms. Intuitively, an invariant of a structure is an "objective" feature of the structure (see [2]). A little reflection shows that the set $\{b,c\}$ is an invariant of the structure in the example above.

Now that we know where we are headed, let us proceed to develop this account of structures within LISP (see Listing IV). First of all, we define a *possible universe* of a structure. The general idea, recall, is that this may be any non-empty set. The predicate **a-universe?**, thus, returns a "true" just in case it is applied to a list that is not the empty list. A **structure?** is now defined as an ordered pair whose first member is a possible universe. Notice that this definition allows the structor of the structure to be the empty list. In this case, the universe would only be distinguished by its cardinality. In the frequently encountered cases where the structor of the universe is a sequence of relations on the universe, such a sequence, recall, is merely a single complex set (a function), and so falls under the definition just provided. For later use, we shall need an operator, **universe-of**, to extract the universe of a structure, and another, **structor-of**, to produce its distinguished structor.

In order to define the automorphisms of a structure, we first need some way to deal with all of the permutations of its universe. The method adopted here is to extract the permutations as needed, rather than first creating a list containing all of them, since their number may be considerable. The procedure **next-perm** takes a list and a permutation of the list as arguments. It then delivers the "next" permutation of the first list. For example, beginning with the arguments $(a\,b\,c\,d)$ and $(a\,b\,c\,d)$, the result is $(a\,b\,d\,c)$. Replacing the second argument by this result, we obtain a third permutation, and so on. When the last permutation is obtained in this way, the empty list is returned. The order of the permutations is probably best seen by running through an example or two. In any case, for our purposes, the order is unimportant; all that matters is

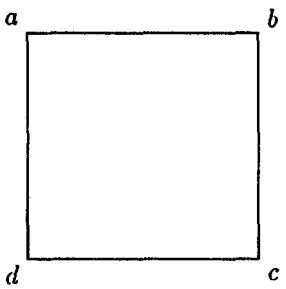
that eventually all the permutations are obtained, and they are. Precisely how the procedure operates is another story.[12]

The automorphisms of a structure are now obtained in the obvious way. The procedure **autos** runs through each permutation of the universe, checking to see if it **preserves?** the distinguished structor. If so, that permutation is added to a running list of automorphisms, which is then returned after all permutations have been examined. An **invariant?** of a structure is any structor of its universe that is preserved by all automorphisms. Often, it is useful to know which subclasses of the universe are invariants, and this may be obtained by using the procedure **invariant-subclasses**. An element of the universe that is mapped to itself by every automorphism is called a *fixed point* of the structure. Of course, some structures may have no fixed points; in any case, the procedure **fixed-point** delivers a list of all of them.

7. SOME EXAMPLES

The procedures of the next section (Listing V) are included in order to facilitate the exploration of sets and structures. The procedure **display-bijection** displays permutations in the more easily readable form of a $2 \times N$ matrix. It is used as an auxiliary to **display-autos**, which prints the automorphisms of the structure to the screen. The procedure **autos-to-printer** outputs the automorphisms in display form to the line printer.

The first example, **dihedral-8**, creates a structure whose universe contains four "points," and whose automorphisms are the symmetries of these four points when they are arranged as a square. These symmetries form a group called the "dihedral group of order 8" (see, e.g., [10, p. 54ff]). In order to generate them, we need to create a structure whose universe contains four points—say, $(a\,b\,c\,d)$—and whose distinguished structor provides the right sort of constraints to generate the various symmetries of a square. To see what this involves, consider the diagram below.



As a first attempt, since the sides of a square are all equal, we might try a function that assigns each side the same value, say one. We then would have the function

$$(\langle\langle a,b\rangle,1\rangle\,\langle\langle b,c\rangle,1\rangle\,\langle\langle c,d\rangle,1\rangle\,\langle\langle d,a\rangle,1\rangle)$$

as our distinguished structor. But clearly this can be simplified, since all that matters here is that the sides be congruent: they need not all be of some *particular* length. If we now identify a side with the set of its two (end) points, then we obtain

$$\text{Sides} = ((a\,b)\,(b\,c)\,(c\,d)\,(d\,a))$$

as our candidate structor. In so doing, we are in effect saying that a permutation of the set $(a\,b\,c\,d)$ will be a symmetry just in case it takes every side to a side. The structure we want to create thus becomes simply

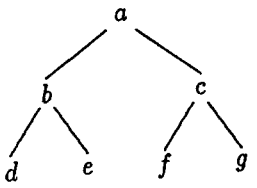
(ordered-pair '(a b c d) Sides),

and this is precisely what the procedure **dihedral-8** delivers. Entering (**display-autos** (**dihedral-8**)) now prints the symmetries of the structure we have created to the screen. Notice that

[12] For the details, see [9, Chapter 7]. The procedures remove and next are auxiliaries used in next-perm; the details of their definitions are routine.

there are eight symmetries, as there ought to be. Furthermore, the diagonals, $(a\,c)$ and $(b\,d)$, are always mapped to each other. In other words, although we did not explicitly require it at the outset, the set of diagonals $((a\,c)\,(b\,d))$ is an invariant of the structure.

The second example, **tree**, is a binary tree containing seven nodes. As before (Section 2), the tree can be pictured as:



and its structure can be represented by the 'parent' relation which holds between a node and the set of its two offspring. Hence, the above tree may be represented by the structure $\langle U, \text{Parent}\rangle$, where

$$U = (a\,b\,c\,d\,e\,f\,g)$$

and

$$\text{Parent} = ((a\,(b\,c))\,(b\,(d\,e))\,(c\,(f\,g))).$$

Entering the procedure **tree** produces this structure, and as before, its automorphisms may now be printed to the screen using **display-autos**, or, better, to the printer, using **autos-to-printer**. (Since there are some 7! permutations to consider, however, the process can take quite some time!) Once they are obtained, it is an easy matter to compare these automorphisms with the figure and verify that they are indeed its symmetries.

8. EXTENSIONS AND REFINEMENTS

As I mentioned earlier, this account has not been geared towards computational efficiency, but conceptual clarity. The execution time of the last example, with its brute force search for automorphisms out of a total of 7! permutations, exhibits this (all too) clearly. Effective computational use of the theory of set-theoretican structures will obviously require extensive modifications of the algorithms used here. While I make no claims to expertise in these matters, let me indicate a few obvious changes that would result in greater computational efficiency.

First of all, there are more efficient ways to compute automorphisms than simply searching the entire permutation group of a structure. The basic idea is that, in some cases, knowing that a permutation is *not* an automorphism may imply that no members of a large set of other permutations are automorphisms either. For example, if a given permutation $p*$ fails to map a distinguished structor to itself simply because it maps, say, a to d and b to g, then any other permutation that delivers these same values for a and b must also fail to be an automorphism. This and related considerations are exploited in an algorithm developed by [11] which is considerably more efficient than brute search.

As defined above, a structure is an ordered pair consisting of a universe and a distinguished structor of that universe. Often, the distinguished structor is simply a sequence of sets. Even when this is the case, however, the above procedure for computing automorphisms will still test a permutation by comparing the entire sequence with its image under the permutation. Sequences being what they are, it clearly suffices to run down the sequence set by set, checking to see whether or not each set is preserved by the permutation under consideration. One way of handling this would be to require at the outset that a structure be an ordered pair consisting of a universe and a *sequence* of structors the universe; the procedure **autos** would then need a simple modification that would have it check each structor in the sequence in succession.

In addition, there are specific features of the *Scheme* dialect that have not been adequately exploited. For example, the **powerset** and **next-perm** procedures could have been written using delayed evaluation, making the later computation of **autos** and **invariant-subclasses** less costly. Finally, it is a simple matter to extend the procedures developed here to allow the computation of the group table of the automorphism group of a structure. Naming the automorphisms and then

using the functional compose relation is all that is required. Notice that once this is done, the resulting set of automorphisms, paired with their group-composition function, forms yet another set-theoretical structure whose automorphisms may be computed in turn.

References

1. J. Winnie, Invariants and objectivity: A theory with applications to relativity and geometry, In *Colodny (1986)*, 71–180 (1986).
2. H. Weyl, *Symmetry*, Princeton University Press, Princeton, New Jersey, (1952).
3. D. Knuth, *The Art of Computer Programming I*, 2nd ed., Addison-Wesley, Reading, Mass., (1973).
4. H. Abelson and J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, Mass., (1985).
5. D. Friedman and M. Felleisen, *The Little LISPer: Trade Edition*, The MIT Press, Cambridge, Mass., (1987).
6. P. Winston and K. Horn, *Lisp*, Addison-Wesley, Reading, Mass., (1984).
7. G. Springer and D. Friedman, *Scheme and the Art of Programming*, The MIT Press, Cambridge, Mass., (1989).
8. W.V. Quine, *Word and Object*, Technology Press & John Wiley & Sons Inc., New York, (1960).
9. A. Nijenhuis and H.S. Wilf, *Combinatorial Algorithms*, Academic Press, Boston, Mass., (1978).
10. I. Grossman and W. Magnus, *Groups and Their Graphs*, Random House, New York, (1964).
11. J.S. Leon, Computing automorphism groups of combinatorial objects, In *Atkinson*, 321–335 (1984).
12. M.D. Atkinson, Ed., *Computational Group Theory*, Academic Press, Boston, (1984).
13. R. Colodny, Ed., *From Quarks to Quasars: Philosophical Problems of Modern Physics*, University of Pittsburgh Press, Pittsburgh, (1986).

APPENDIX LISTINGS I–V

LISTING I. Logical Operators

```
(define some?                    ;Does some item in l have the property?
  (lambda (property l)           ;If so, return 'true. (Not  the first
   (cond                         ;such item, for it could be '()!).
    ((null? l) nil)
    ((property (car l)) t)
    (else (some? property (cdr l))))))

(define all?                     ;Does every item in l have the property?
  (lambda (property l)
   (cond
    ((null? l) t)
    ((property (car l)) (all? property (cdr l)))
    (else nil))))

(define none?                     ;Do no items in l have the property?
  (lambda (property l)
    (not (some? property l))))

(define one-every?                ;Does e have the relation to every item in l?
  (lambda (e relation l)
    (all? (lambda (x) (relation e x)) l)))

(define every-one?                ;Does every item in l have the relation to e?
  (lambda (l relation e)
    (all? (lambda (x) (relation x e)) l)))

(define one-some?                 ;Does e have the relation to some item in l?
  (lambda (e relation l)
    (some? (lambda (x) (relation e x)) l)))

(define every-some?                ;Does every item in l have the relation
  (lambda (l relation m)           ;to some item in m?
    (all? (lambda (x) (one-some? x relation m)) l)))
```

LISTING II. Fundamental Set-Theoretical Operations

```
(define same-set?                    ;Are l and m the same set?
  (lambda (l m)
   (cond
     ((eqv? l m) t)
     ((or (atom? l) (atom? m)) nil)
     (else (and (subset? l m) (subset? m l))))))

(define subset?                      ;Is l a subset of m?
  (lambda (l m)
    (every-some? l same-set? m)))

(define element?                     ;Is e an element of set l?
  (lambda (e l)
   (one-some? e same-set? l)))

(define powerset                     ;Returns the set of all subsets of l.
  (lambda (l)
   (cond
    ((null? l) (list nil))
    (else (append (map (lambda (x) (cons (car l) x))
                       (powerset (cdr l)))
                  (powerset (cdr l)))))))

(define cardinal                     ;Returns the number of elements in the set l.
  (lambda (l)
    (cond
      ((null? l) 0)
      ((element? (car l) (cdr l)) (cardinal (cdr l)))
      (else (add1 (cardinal (cdr l)))))))

(define set-union                    ;Returns the set of all items in l or in m.
  (lambda (l m)
    (cond
      ((or (atom? l) (null? l)) m)
      ((or (atom? m) (null? m)) l)
      ((element? (car l) m) (set-union (cdr l) m))
      (else (cons (car l) (set-union (cdr l) m))))))

(define set-intersection             ;Returns the set of items in both l and m.
  (lambda (l m)
   (cond
    ((or (null? l) (null? m) (atom? l) (atom? m)) nil)
    ((element? (car l) m) (cons (car l) (set-intersection (cdr l) m)))
    (else (set-intersection (cdr l) m)))))

(define set-difference               ;Returns the set of items in l that are
  (lambda (l m)                      ;not in m.
   (cond
    ((null? m) l)
    ((null? l) nil)
    ((element? (car l) m) (set-difference (cdr l) m))
    (else (cons (car l) (set-difference (cdr l) m))))))

;Separation.

(define separate                     ;Returns the set of all items in l having
  (lambda (property l)               ;the property.
   (cond
    ((null? l) nil)
    ((property (car l)) (cons (car l) (separate property (cdr l))))
    (else (separate property (cdr l))))))

;Operations on families of sets (sets of sets).

(define union-of-sets                ;Yields the union of all sets in l.
  (lambda (l)
   (cond
    ((null? l) nil)
    (else (set-union (car l) (union-of-sets (cdr l)))))))
```

```
(define intersection-of-sets     ;Yields the intersection of all sets in l.
  (lambda (l)
   (cond
    ((null? l) nil)
    ((null? (cdr l)) (car l))
    (else (set-intersection (car l) (intersection-of-sets (cdr l)))))))

(define collapse      ;Yields the set l purged of duplicates at all levels.
  (lambda (l)
    (cond
      ((or (null? l) (atom? l)) l)
      ((element? (car l) (cdr l)) (collapse (cdr l)))
      (else (cons (collapse (car l)) (collapse (cdr l)))))))
```

LISTING III. Ordered Pairs, Functions, and Images

```
(define ordered-pair                    ;Yields ((x) (x y))--the Kuratowski
  (lambda (x y)                         ;definition (see text).
   (list (list x) (list x y))))

(define ordered-pair?   ;Is l an ordered-pair? That is, a non-empty family
  (lambda (l)           ;of sets whose intersection contains exactly one
    (cond               ;element, and whose union contains at most two?
      ((null? l) '())
      ((some? atom? l) '())
      (else (and (= (cardinal (intersection-of-sets l)) 1)
                 (<= (cardinal (union-of-sets l)) 2))))))

(define first-of-pair               ;Returns the first member of l.
  (lambda (l)
    (car (intersection-of-sets l))))

(define second-of-pair              ;Returns the second member of l.
  (lambda (l)
    (let ((diff (set-difference (union-of-sets l) (intersection-of-sets l))))
    (if (null? diff) (first-of-pair l) (car diff)))))

(define cross-product               ;Returns the set of all ordered pairs
  (lambda (l m)                     ;having first elements in l and second
    (cond                           ;elements in m.
      ((or (null? l) (null? m)) '())
      (else (append (mapcar (lambda (x) (ordered-pair (car l) x)) m)
                    (cross-product (cdr l) m))))))

;Functions and Related Concepts.

(define create-function                 ;Given two lists, the result is a
  (lambda (l m)                         ;function that takes eachitem in
    (cond                               ;the first list to the corresponding
      ((or (null? l) (null? m)) '())    ;item in the second.
      (else (cons (ordered-pair (car l) (car m))
                  (create-function (cdr l) (cdr m)))))))

(define domain                          ;Returns the set of objects on which
  (lambda (funct)                       ;the function is defined.
    (mapcar first-of-pair funct)))

(define range                 ;Returns the image of the domain under function
  (lambda (funct)             ;funct.  Note that the range is collapsed.
    (collapse (mapcar second-of-pair funct))))

(define value               ;Returns the value of funct on the argument arg,
  (lambda (funct arg)       ;or, if arg is not in the domain, an error.
    (cond
      ((null? funct) (error "Argument not in domain of function."))
      ((same-set? (first-of-pair (car funct)) arg) (second-of-pair (car funct)))
      (else (value (cdr funct) arg)))))

;Sequences.
```

```
(define create-sequence          ;When l = (a b c ...), the result is
  (lambda (l)                    ;the sequence (<0,a> <1,b> <2,c> ...).
    (do ((l l (cdr l)) (count 0 (add1 count)) (seq '()))
        ((null? l) (reverse seq))   ;exit.
        (set! seq (cons (ordered-pair count (car l)) seq)))))

;Images.

(define image-aux                ;Returns the image of the target set under
  (lambda (funct domain target) ;the function. See text for details.
    (cond
      ((element? target domain) (value funct target))
      ((atom? target) target)
      (else (mapcar (lambda (x) (image-aux funct domain x)) target)))))

(define image                    ;Returns the image of the target set under
  (lambda (funct target)         ;the function.
    (image-aux funct (domain funct) target)))

(define compose                   ;Returns fun1 composed with fun2.
  (lambda (fun1 fun2)
    (mapcar (lambda (x) (ordered-pair x (value fun1 (value fun2 x))))
            (domain fun2))))

(define preserves?   ;Is e left untouched by the function?
  (lambda (funct e)
    (same-set? (image funct e) e)))
```

Listing IV. Set-Theoretical Structures and Symmetries

```
(define a-universe?                          ;Is x a non-empty list?
  (lambda (x)
    (and (not (atom? x)) (not (null? x)))))

(define structure?          ;Is the set an ordered pair whose first member is a
  (lambda (s)               ;possible universe?
    (and (ordered-pair? s)
         (a-universe? (first-of-pair s)))))

(define universe-of         ;Returns the universe of a structure.
  (lambda (s)
    (first-of-pair s)))

(define structor-of        ;Returns the distinguished structor of a structure.
  (lambda (s)
    (second-of-pair s)))

;Auxiliaries needed for Automorphisms: Permutations.

(define remove      ;Return the result of removing the first occurence
  (lambda (e l)     ;of item e from the list l.
    (if (equal? e (car l)) (cdr l) (cons (car l) (remove e (cdr l))))))

(define next        ;Returns the entry immediately following e in the list l.
  (lambda (l e)     ;If, for any reason, there is no such entry, '() is
    (cond           ;returned.
      ((null? (cdr l)) '())
      ((equal? (car l) e) (cadr l))
      (else (next (cdr l) e)))))

(define next-perm       ;Returns a list that is the permutation of start
  (lambda (start perm) ;following permutation l.  Thus,
    (cond               ;(next-perm '(a b c d) '(a b d c)) returns '(a c b d).
      ((null? perm) '())  ;When perm is the last permutation, return '().
      ((next-perm (remove (car perm) start) (cdr perm))
       (cons (car perm) (next-perm (remove (car perm) start) (cdr perm))))
      ((next start (car perm))
       (cons (next start (car perm)) (remove (next start (car perm)) start)))
      (else '()))))
```

```
;Automorphisms (Symmetries) and Invariants.

(define autos           ;Returns the set of all automorphisms (symmetries)
  (lambda (s)           ;of the structure s.
    (let ((univ (collapse (universe-of s)))
          (struct (collapse (structor-of s))))
     (do ((perm '())
          (auts (list (create-function univ univ)))  ;Put identity map in auts.
          (newperm (next-perm univ univ)             ;Start with next perm.
                   (next-perm univ newperm)))
          ((null? newperm) auts)                     ;All perms in? Return auts.
          (set! perm (create-function univ newperm)) ;Make the perm function.
          (if (preserves? perm struct)               ;Does perm preserve the
              (set! auts (cons perm auts)))))))      ;structors? If so, add it.

(define invariant?        ;Is the structor l an invariant of structure s?
  (lambda (l s)
    (every-one? (autos s) preserves? l)))

(define invariant-subclasses  ;Returns the class of all invariant subclasses
  (lambda (s)                 ;of the universe of structure s.
    (let ((auts (autos s)))
         (separate (lambda (x) (every-one? auts preserves? x))
                   (powerset (universe-of s))))))

(define fixed-points          ;Returns the class of all fixed points of the
  (lambda (s)                 ;structure s.
    (let ((auts (autos s)) (univ (universe-of s)))
         (separate (lambda (x) (every-one? auts preserves? x)) univ))))
```

Listing V. Display Functions and Examples

```
; Displaying bijections

(define display-bijection           ;Prints a bijection in permutation form.
  (lambda (f)
    (print (domain f))
    (print (range f))
    (newline)))

(define display-autos               ;Outputs the automorphisms of a structure
  (lambda (s)                       ;s in display form to the console.
    (mapc display-bijection (autos s))))

(define autos-to-printer             ;Outputs the automorphisms of a structure
  (lambda (s)                        ;s in display form to the line printer.
    (let ((c (current-output-port))
          (p (open-output-file "prn")))
      (set-fluid! output-port p)
      (display-autos s)
      (close-output-port p)
      (set-fluid! output-port c))))

; Examples

(define dihedral-8
  (lambda ()
    (ordered-pair '(a b c d) '((a b) (b c) (c d) (d a)))))

(define tree
  (lambda ()
    (let ((u '(a b c d e f g))
          (struct (list '(a (b c)) '(b (d e)) '(c (f g)))))
    (ordered-pair u struct))))
```